Last Class: CPU Scheduling

- **Scheduling Algorithms:**
  - FCFS
  - Round Robin
  - SJF
  - Multilevel Feedback Queues
  - Lottery Scheduling
- **Review questions:**
  - How does each work?
  - Advantages? Disadvantages?

Today: Synchronization

- What kind of knowledge and mechanisms do we need to get independent processes to communicate and get a consistent view of the world (computer state)?
- **Example: Too Much Milk**

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

Synchronization Terminology

- **Synchronization:** use of atomic operations to ensure cooperation between threads
- **Mutual Exclusion:** ensure that only one thread does a particular activity at a time and *excludes* other threads from doing it at that time
- **Critical Section:** piece of code that only one thread can execute at a time
- **Lock:** mechanism to prevent another process from doing something
  1. Lock before entering a critical section, or before accessing shared data.
  2. Unlock when leaving a critical section or when access to shared data is complete
  3. Wait if locked

⇒ All synchronization involves waiting.

Too Much Milk: Solution 1

- What are the correctness properties for this problem?
  - Only one person buys milk at a time.
  - Someone buys milk if you need it.
- Restrict ourselves to atomic loads and stores as building blocks.
  - Leave a note (a version of lock)
  - Remove note (a version of unlock)
  - Do not buy any milk if there is note (wait)

```
Thread A                      Thread B

if (noMilk & NoNote) {        if (noMilk & NoNote) {
    leave Note;                   leave Note;
    buy milk;                     buy milk;
    remove note;                 remove note;
}                             }
Does this work?
```

Too Much Milk: Solution 2

How about using labeled notes so we can leave a note before checking the the milk?

```
Thread A                    Thread B

leave note A                leave note B
if (noNote B) {             if (noNote A) {
   if (noMilk){                if (noMilk){
      buy milk;                   buy milk;
   }                           }
}                            }
remove note;                 remove note;
```

Does this work?

Too Much Milk: Solution 3

```
   Thread A                   Thread B

   leave note A               leave note B
X: while (Note B) {        Y: if (noNote A) {
      do nothing;                 if (noMilk){
   }                                 buy milk;
   if (noMilk){                   }
       buy milk;              }
   }                          remove note B;
   remove note A;
```

Does this work?

Correctness of Solution 3

- At point Y, either there is a note A or not.
  1. If there is no note A, it is safe for thread B to check and buy milk, if needed. (Thread A has not started yet).
  2. If there is a note A, then thread A is checking and buying milk as needed or is waiting for B to quit, so B quits by removing note B.
- At point X, either there is a note B or not.
  1. If there is not a note B, it is safe for A to buy since B has either not started or quit.
  2. If there is a note B, A waits until there is no longer a note B, and either finds milk that B bought or buys it if needed.
- Thus, thread B buys milk (which thread A finds) or not, but either way it removes note B. Since thread A loops, it waits for B to buy milk or not, and then if B did not buy, it buys the milk.

Is Solution 3 a good solution?

1. It is too complicated - it was hard to convince ourselves this solution works.
2. It is asymmetrical - thread A and B are different. Thus, adding more threads would require different code for each new thread and modifications to existing threads.
3. A is *busy waiting* - A is consuming CPU resources despite the fact that it is not doing any useful work.

⇒This solution relies on loads and stores being atomic.

Language Support for Synchronization

Have your programming language provide atomic routines for synchronization.

- **Locks:** one process holds a lock at a time, does its critical section releases lock.
- **Semaphores:** more general version of locks.
- **Monitors:** connects shared data to synchronization primitives.

⇒ All of these require some hardware support, and waiting.

Locks

- **Locks:** provide mutual exclusion to shared data with two "atomic" routines:
  - **Lock::Acquire** - wait until lock is free, then grab it.
  - **Lock::Release** - unlock, and wake up any thread waiting in Acquire.

Rules for using a lock:

- Always acquire the lock before accessing shared data.
- Always release the lock after finishing with shared data.
- Lock is initially free.

Implementing Too Much Milk with Locks

Too Much Milk

```
Thread A                    Thread B

Lock->Acquire();            Lock->Acquire();
if (noMilk){                if (noMilk){
   buy milk;                   buy milk;
}                           }
Lock->Release();            Lock->Release();
```

- This solution is clean and symmetric.
- How do we make Lock::Acquire and Lock::Release atomic?

Hardware Support for Synchronization

- Implementing high level primitives requires low-level hardware support
- What we have and what we want

	Concurrent Programs
Low-level atomic operations (hardware)	load/store, interrupt disable, test&set
High-level atomic operations (software)	locks, semaphores, monitors, send & receive

Implementing Locks By Disabling Interrupts

- There are two ways the CPU scheduler gets control:
  - **Internal Events:** the thread does something to relinquish control (e.g., I/O).
  - **External Events:** interrupts (e.g., time slice) cause the scheduler to take control away from the running thread.
- On uniprocessors, we can prevent the scheduler from getting control as follows:
  - **Internal Events:** prevent these by not requesting any I/O operations during a critical section.
  - **External Events:** prevent these by disabling interrupts (i.e., tell the hardware to delay handling any external events until after the thread is finished with the critical section)
- Why not have the OS support Lock::Acquire() and Lock::Release as system calls?

Implementing Locks by Disabling Interrupts

- For uniprocessors, we can disable interrupts for high-level primitives like locks, whose implementations are private to the kernel.
- The kernel ensures that interrupts are not disabled forever, just like it already does during interrupt handling

```
class Lock {
  public:
    void Acquire();
    void Release();
  private:
    int value;
    Queue Q;
}
Lock::Lock {
  // lock is free
  value = 0;
  // queue is empty
  Q = 0;
}
```

```
Lock::Acquire(T:Thread){
  disable interrupts;
  if (value == BUSY) {
     add T to Q
     T->Sleep();
  } else {
     value = BUSY;
  }
  enable interrupts; }
```

```
Lock::Release() {
  disable interrupts;
  if queue not empty {
     take thread T off Q
     put T on ready queue
  } else {
     value = FREE
  }
  enable interrupts; }
```

Wait Queues

When should Acquire re-enable interrupts when going to sleep?

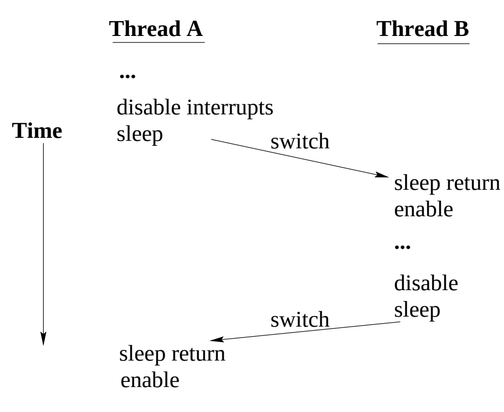
- **Before putting the thread on the wait queue?** No, Release could check the queue, and not wake up the thread.
- **After putting the thread on the wait queue, but before going to sleep?** No, Release could put the thread on the ready queue, but it could already be on the ready queue. When the thread wakes up, it will go to sleep, missing the wakeup from Release.

⇒ In Nachos, Thread::Sleep() disables interrupts and it is the responsibility of the next thread that executes to enable interrupts. (The dispatcher can enable interrupts every time it selects a new process to run.)

We still have a problem with multiprocessors.

Example

- When the sleeping thread wakes up, it returns from Sleep back to Acquire.
- Interrupts are still disabled, so its ok to check the lock value, and if it is free, grab the lock and turn on interrupts.



Summary

- Communication among threads is typically done through shared variables.
- Critical sections identify pieces of code that cannot be executed in parallel by multiple threads, typically code that accesses and/or modifies the values of shared variables.
- Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.
  - Achieving synchronization directly with loads and stores is tricky and error-prone
  - *Solution:* use high-level primitives such as locks, semaphores, monitors